



CUDA

Prof. Michele Amoretti

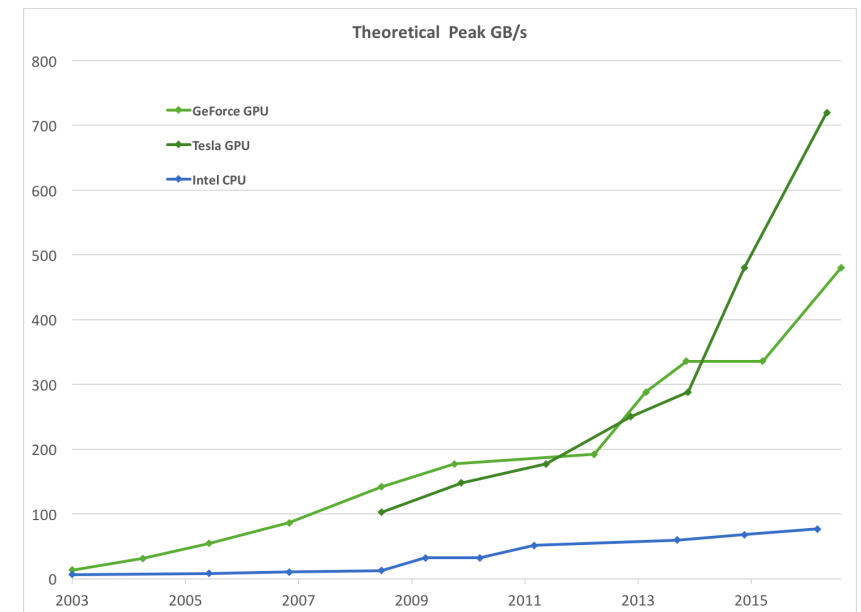
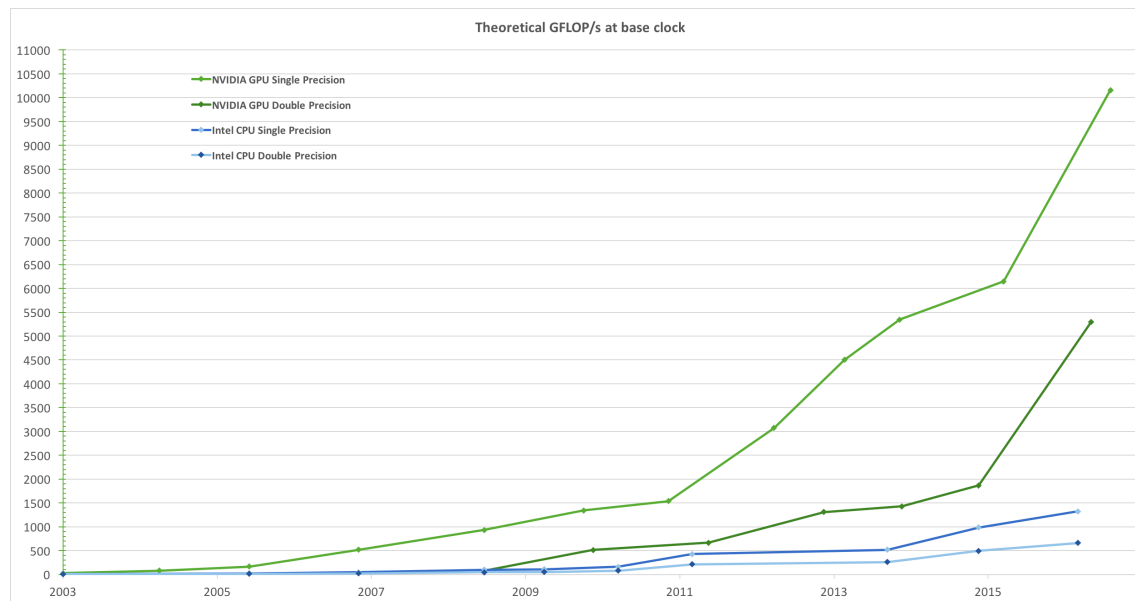
High Performance Computing 2022/2023



Introduction

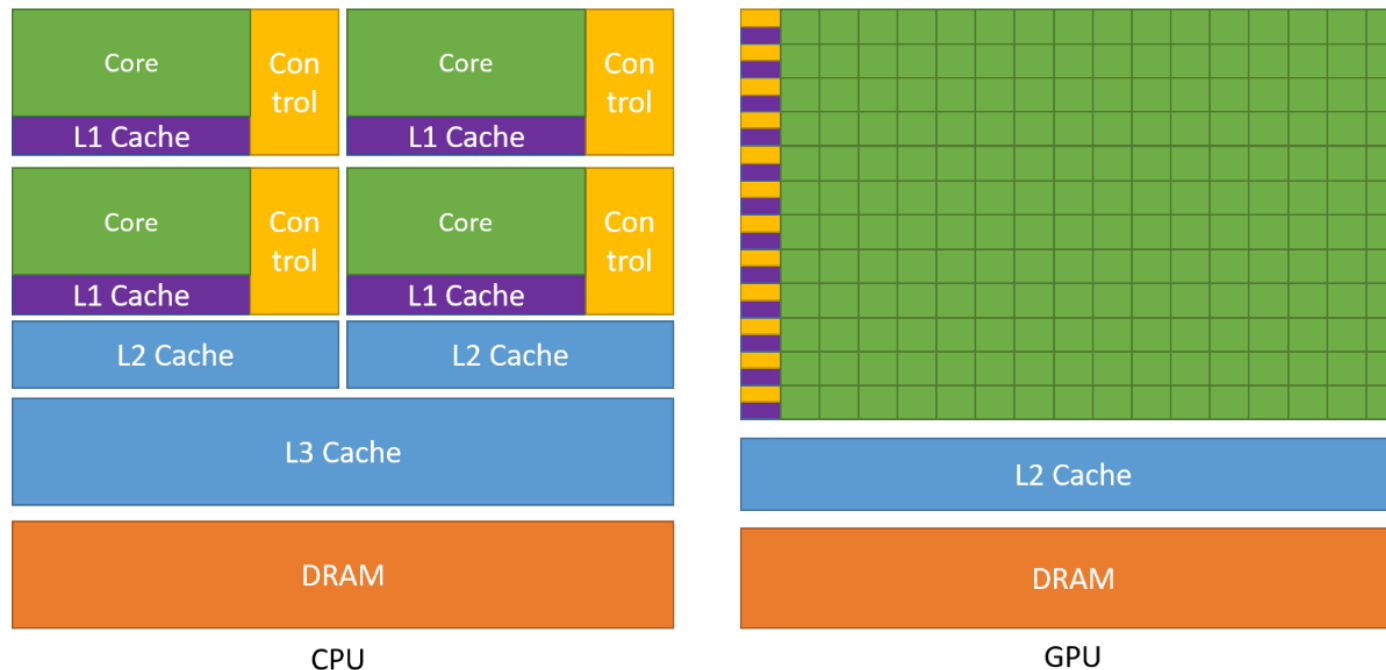
From graphics processing to general-purpose parallel computing

Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable **Graphic Processor Unit (GPU)** has evolved into a highly parallel, multithreaded, **manycore processor** with tremendous computational horsepower and very high memory bandwidth.



From graphics processing to general-purpose parallel computing

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for **compute-intensive, highly parallel computation** – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.



From graphics processing to general-purpose parallel computing

GPUs are especially well-suited to address problems that

- can be expressed as data-parallel computations – the same program is executed on many data elements in parallel
- with high arithmetic intensity – the ratio of arithmetic operations to memory operations



- lower requirement for sophisticated flow control
- no need for large caches






Data-parallel processing maps data elements to **parallel processing threads**. Many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

CUDA: a general-purpose parallel computing architecture

Introduced by NVIDIA in 2006.

CUDA comes with a software environment (**CUDA Toolkit**) that allows developers to use C/C++ as a high-level programming language.

Other languages or application programming interfaces are also supported, such as FORTRAN, OpenACC, DirectCompute.

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
<div>CUDA-Enabled NVIDIA GPUs</div>						
NVIDIA Ampere Architecture (compute capabilities 8.x)					Tesla A Series	
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series		Tesla T Series	
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series		Tesla V Series	
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series		Tesla P Series	
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center		

To access CUDA from Python: <https://developer.nvidia.com/cuda-python>
 Java bindings for CUDA: <http://javagl.de/jcuda.org/>



Downloads

NVIDIA's CUDA development tools:

1. Latest CUDA driver
2. CUDA Toolkit
3. CUDA code samples

<https://developer.nvidia.com/cuda-downloads>

CUDA Toolkit

The CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications.

The CUDA Toolkit includes a compiler for NVIDIA GPUs (**nvcc**), math libraries, and a tool for debugging and optimizing the performance of the applications (**cuda-gdb**).

There are also programming guides, user manuals, API reference, and other documentation to help the user get started quickly.

nvcc

nvcc is a compiler that provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages.

Source files can include a mix of **host code** (i.e., code that executes on the host) and **device code** (i.e., code that executes on the device).

nvcc's basic workflow consists in separating device code from host code and compiling the device code into an assembly form (*PTX code*) and/or binary form (*cubin object*).

nvcc

The front end of the compiler processes CUDA source files according to C++ syntax rules.

C++ is supported for the host code and (starting from CUDA 7.0) for the device code (with some restrictions for C++ functions: can only access GPU memory, no variable number of arguments, no static variables).

For details, refer to:

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-cplusplus-language-support>

Thrust is a C++ template library for CUDA based on the Standard Template Library (STL). Thrust allows you to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C.

<https://docs.nvidia.com/cuda/thrust/index.html>



Programming model

CUDA's scalable programming model

The challenge is to develop **application software that transparently scales its parallelism to leverage the increasing number of processor cores**, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

CUDA's parallel programming model is designed to overcome this challenge while being simple for programmers that are familiar with standard programming languages such as C.

CUDA's scalable programming model

At its core are three key abstractions:

- **a hierarchy of thread groups**
- **shared memories**
- **barrier synchronization**

that are simply exposed to the programmer as a minimal set of language extensions.

These abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel.

A compiled CUDA program can therefore execute on any number of processor cores, and only the runtime system needs to know the physical processor count.

Programming APIs

Two interfaces are currently supported to write CUDA programs:

- ❑ **CUDA C** (a minimal set of extensions to the C language)
- ❑ **CUDA driver API** (a low-level C API)

They are mutually exclusive: a program must use either one or the other.

CUDA C comes with a runtime API, which is built on top of the CUDA driver API.

Both the runtime API and the driver API provide functions to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc.

Programming APIs

Initialization, context, and module management are all implicit and resulting code is more concise.

In contrast, the CUDA driver API requires more code, is harder to program and debug, but offers a better level of control and is language-independent since it handles binary or assembly code.

**In this presentation we focus on CUDA C
(the current version of the software platform is 11.6).**

Kernels

CUDA C extends C by allowing the programmer to define C functions, denoted as **kernels**, that are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads for each call is specified using a new `<<<...>>>` syntax:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```


Kernels

Each thread that executes a kernel is given a thread index that is accessible within the kernel through the built-in **threadIdx** variable. As an illustration, the following sample code adds two vectors A and B of size N and stores the result into vector C :

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>>(A, B, C);
    ...
}
```

Each thread that executes **VecAdd()** performs one pair-wise sum.

Thread hierarchy

For convenience, **threadIdx** is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional **thread block**.

This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or field.

As an example, the following code adds two matrices A and B of size $N \times N$ and stores the result into matrix C :

Thread hierarchy

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread hierarchy

threadID is a unique, scalar number that identifies each thread uniquely in a threadblock regardless of whether that threadblock is 1D, 2D or 3D.

The thread index and the threadID relate to each other in a straightforward way:

- for a one-dimensional block, they are the same;
- for a two-dimensional block of size (Dx, Dy) , the threadID of a thread of index (x, y) is $(x + y Dx)$;
- for a three-dimensional block of size (Dx, Dy, Dz) , the threadID of a thread of index (x, y, z) is $(x + y Dx + z Dx Dy)$.

In this way, it is possible to order threads by increasing ID.

Thread hierarchy

Threads within a **block** can cooperate by sharing data through some per-block **shared memory** and synchronizing their execution to coordinate memory accesses.

One can specify synchronization points in the kernel by calling the **__syncthreads()** intrinsic function; **__syncthreads()** acts as a **barrier** at which all threads in the block must wait before any is allowed to proceed.

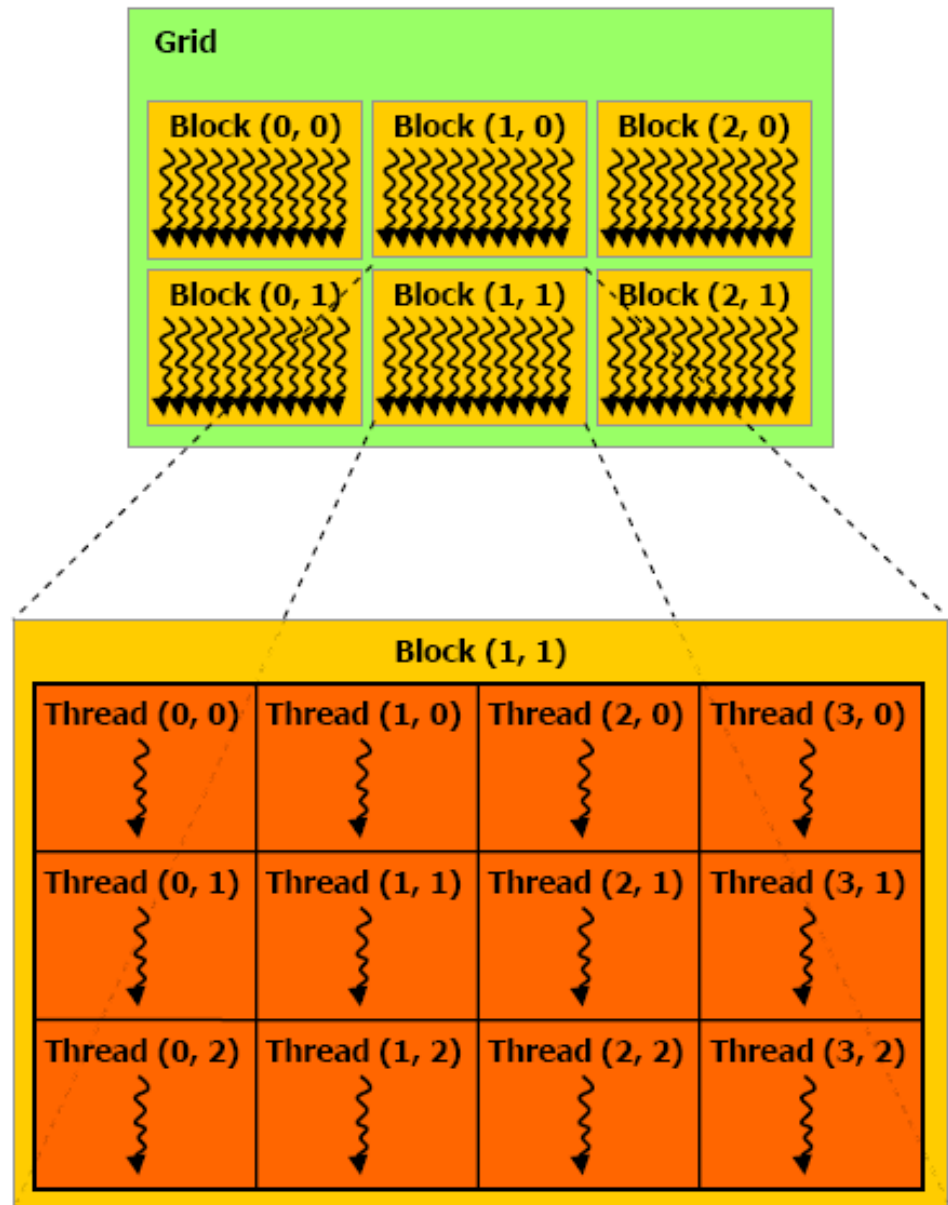
For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core, much like an L1 cache, **__syncthreads()** is expected to be lightweight, and all threads of a block are expected to reside on the same processor core. The number of threads per block is therefore restricted by the limited memory resources of a processor core. **On current GPUs, a thread block may contain up to 1024 threads.**

Thread hierarchy

A kernel can be executed by multiple equally-shaped thread blocks, so that

total number of threads
=
number of threads per block
×
number of blocks

These multiple blocks are organized into a one-dimensional or two-dimensional **grid of thread blocks**.



Thread hierarchy

The dimension of the grid is specified by the first parameter of the `<<<...>>>` syntax.

Each block within the grid can be identified by a one-dimensional or two-dimensional index accessible within the kernel through the built-in **blockIdx** variable.

The dimension of the thread block is accessible within the kernel through the built-in **blockDim** variable. The previous sample code becomes:

Thread hierarchy

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```


Thread hierarchy

A thread block size of 16×16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before.

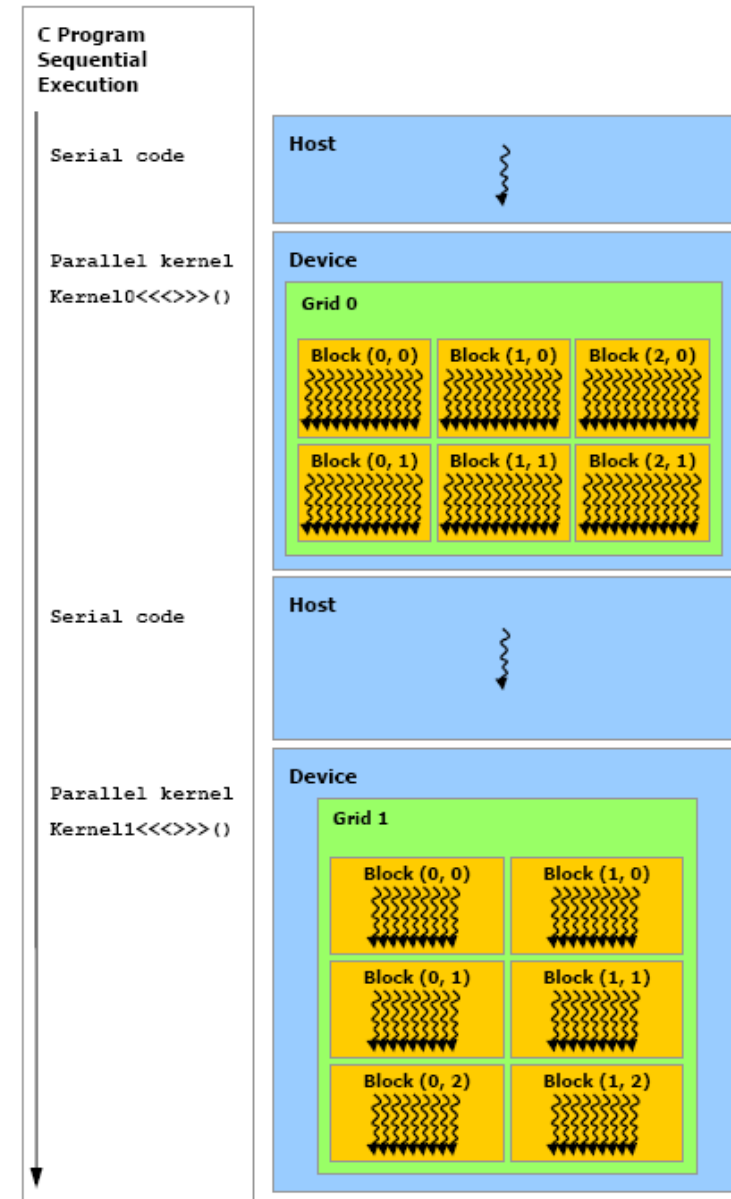
Thread blocks are required to execute independently: it must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores.

The number of thread blocks in a grid is typically dictated by the size of the data being processed rather than by the number of processors in the system, which it can greatly exceed.

Host and device

CUDA's programming model assumes that CUDA threads execute on **a physically separate device**, which operates as a coprocessor to the **host** that runs the C program.

This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.





Host and device

CUDA's programming model also assumes that both the host and the device maintain their own DRAM:

host memory

and

device memory

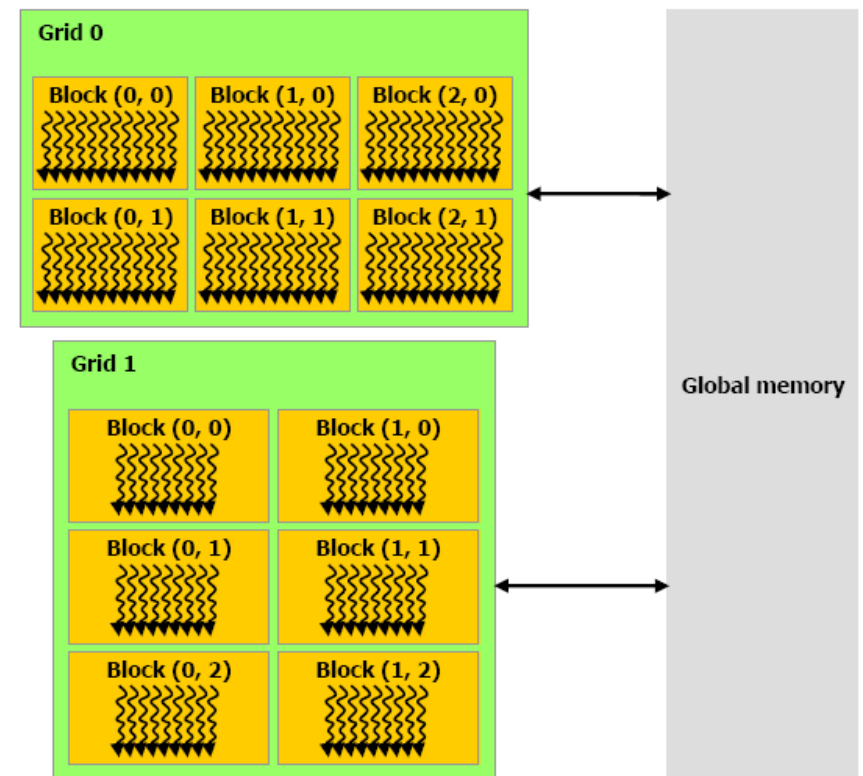
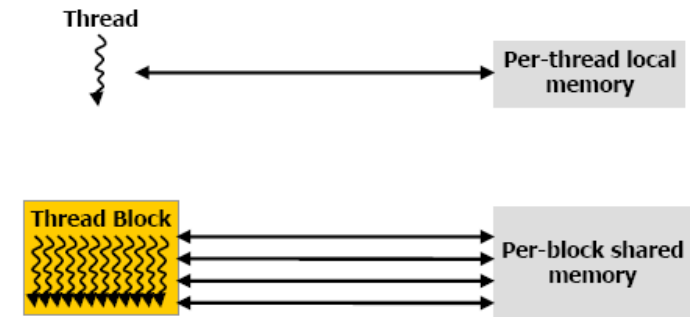
Device memory

CUDA threads may access data from multiple memory spaces during their execution.

Each thread has a private local memory (max 255 registers).

Each thread block has a **shared memory** (64 KB) visible to all threads of the block and with the same lifetime as the block.

Finally, all threads have access to the same **global memory** (12-40 GB).



Device memory

There are also two additional read-only memory spaces accessible by all threads: the **constant** and **texture** memory spaces.

The global, constant, and texture memory spaces are optimized for different memory usages.

Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats.

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

Global memory

Global memory is typically allocated using **cudaMalloc()** and freed using **cudaFree()**.

Data transfer between host memory and global memory are typically performed using **cudaMemcpy()**.

Global memory

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);
}
```

Global memory

```
// Copy vectors from host memory to device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid =
    (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

// Free host memory
...
}
```


Asynchronous concurrent execution

The following device operations are asynchronous with respect to the host:

- ▶ Kernel launches
- ▶ Memory copies within a single device's memory
- ▶ Memory copies from host to device of a memory block of 64 KB or less
- ▶ Memory copies performed by functions that are suffixed with **Async**
- ▶ Memory set function calls

Error handling

All runtime functions return an **error code**, but for an asynchronous function, this error code cannot possibly report any of the asynchronous errors that could occur on the device since the function returns before the device has completed the task.

The error code only reports errors that occur on the host prior to executing the task, typically related to parameter validation; if an asynchronous error occurs, it will be reported by some subsequent unrelated runtime function call.

The only way to check for asynchronous errors just after some asynchronous function call is therefore to synchronize just after the call by calling **cudaDeviceSynchronize()** (or by using any other synchronization mechanisms) and checking the error code returned by that function.

Error handling

The runtime maintains an error variable for each host thread that is initialized to **cudaSuccess** and is overwritten by the error code every time an error occurs (be it a parameter validation error or an asynchronous error).

cudaPeekAtLastError() returns this variable

cudaGetLastError() returns this variable and resets it to **cudaSuccess**

Dynamic Parallelism

Dynamic Parallelism is an extension to the CUDA programming model **enabling a CUDA kernel to create and synchronize with new work directly on the GPU.**

The ability to create work directly from the GPU can **reduce the need to transfer execution control and data between host and device**, as launch configuration decisions can now be made at runtime by threads executing on the device.

Dynamic Parallelism is only supported by devices of **compute capability 3.5 and higher.**

Dynamic Parallelism

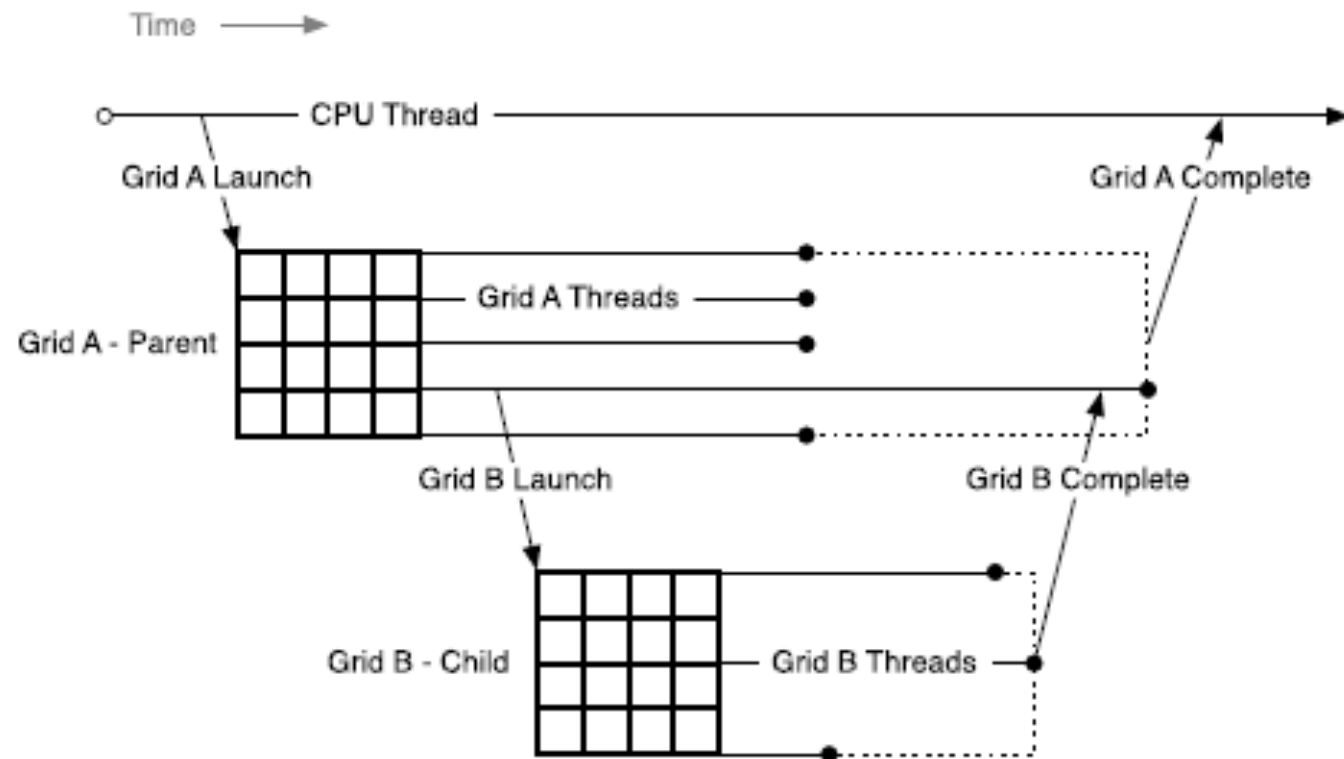
Additionally, **data-dependent parallel work can be generated inline within a kernel at run-time**, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads.

Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed.

Dynamic Parallelism

A device thread that configures and launches a new grid belongs to the **parent grid**, and the grid created by the invocation is a **child grid**.

The invocation and completion of child grids is properly nested, meaning that the parent grid is not considered complete until all child grids created by its threads have completed, and the runtime guarantees an implicit synchronization between the parent and child.



Dynamic Parallelism

CUDA runtime operations from any thread, including kernel launches, are visible across all the threads in a grid.

Execution of a grid is not considered complete until all launches by all threads in the grid have completed.

If all threads in a grid exit before all child launches have completed, an implicit synchronization operation will automatically be triggered.

Dynamic Parallelism

CUDA Streams and **Events** allow control over dependencies between grid launches: **grids launched into the same stream execute in-order**, and events may be used to create dependencies between streams. Streams and events created on the device serve this exact same purpose.

Within a grid, all kernel launches into the same stream (with the exception of the fire-and-forget stream discussed later) are executed in-order. With multiple threads in the same grid launching into the same stream, the ordering within the stream is dependent on the thread scheduling within the grid, which may be controlled with synchronization primitives such as `__syncthreads()`.

Dynamic Parallelism

The device runtime is a functional subset of the host runtime. API level device management, kernel launching, device memcpy, stream management, and event management are exposed from the device runtime.

Programming for the device runtime should be familiar to someone who already has experience with CUDA. Device runtime syntax and semantics are largely the same as that of the host API, with any exceptions detailed earlier in this document.

Dynamic Parallelism

Example

```
#include <stdio.h>

__global__ void childKernel()
{
    printf("Hello ");
}

__global__ void tailKernel()
{
    printf("World!\n");
}

...
```

Dynamic Parallelism

...

```
__global__ void parentKernel()  
{  
    // launch child  
    childKernel<<<1,1>>>();  
    if (cudaSuccess != cudaGetLastError()) {  
        return;  
    }  
  
    // launch tail into cudaStreamTailLaunch stream  
    // implicitly synchronizes: waits for child to complete  
    tailKernel<<<1,1,0,cudaStreamTailLaunch>>>();  
}
```

...

Dynamic Parallelism

...

```
int main(int argc, char *argv[])
{
    // launch parent
    parentKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return 1;
    }

    // wait for parent to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return 2;
    }

    return 0;
}
```

Dynamic Parallelism

A device runtime program may be compiled and linked in a single step, if all required source files can be specified from the command line:

```
$ nvcc -arch=sm_75 -rdc=true hello_world.cu -o hello -lcudadevrt
```

It is also possible to compile CUDA .cu source files first to object files, and then link these together in a two-stage process:

```
$ nvcc -arch=sm_75 -dc hello_world.cu -o hello_world.o  
$ nvcc -arch=sm_75 -rdc=true hello_world.o -o hello -lcudadevrt
```

Dynamic Parallelism

Parent and child grids share the same global and constant memory storage, but have distinct local and shared memory.

There is only one point of time in the execution of a child grid when its view of memory is fully consistent with the parent thread: at the point when the child grid is invoked by the parent.

All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid.

The only way, for the parent grid, to access the modifications made by the threads in the child grid is via a kernel launched into the `cudaStreamTailLaunch` stream.

Dynamic Parallelism

Example

```
__global__ void tail_launch(int *data) {  
    data[threadIdx.x] = data[threadIdx.x]+1;  
}  
  
__global__ void child_launch(int *data) {  
    data[threadIdx.x] = data[threadIdx.x]+1;  
}  
  
__global__ void parent_launch(int *data) {  
    data[threadIdx.x] = threadIdx.x;  
  
    __syncthreads();  
  
    if (threadIdx.x == 0) {  
        child_launch<<< 1, 256 >>>(data);  
        tail_launch<<< 1, 256, 0, cudaStreamTailLaunch >>>(data);  
    }  
}  
  
void host_launch(int *data) {  
    parent_launch<<< 1, 256 >>>(data);  
}
```

The child grid executing `child_launch` is only guaranteed to see the modifications to `data` made before the child grid was launched. Since thread 0 of the parent is performing the launch, the child will be consistent with the memory seen by thread 0 of the parent. Due to the first `__syncthreads()` call, the child will see `data[0]=0`, `data[1]=1`, ..., `data[255]=255`.

Dynamic Parallelism

Shared and Local memory is private to a thread block or thread, respectively, and is not visible or coherent between parent and child. Behavior is undefined when an object in one of these locations is referenced outside of the scope within which it belongs, and may cause an error.

The NVIDIA compiler will attempt to warn if it can detect that a pointer to local or shared memory is being passed as an argument to a kernel launch. At runtime, the programmer may use the `__isGlobal()` intrinsic to determine whether a pointer references global memory and so may safely be passed to a child launch.

Unified Memory

Unified Memory is a component of the CUDA programming model, first introduced in CUDA 6.0, that defines a **managed memory space** in which all processors (CPUs and GPUs) see a single coherent memory image with a common address space.

The underlying system manages data access and locality within a CUDA program without need for explicit memory copy calls. This benefits GPU programming in two primary ways:

- **GPU programming is simplified** by unifying memory spaces coherently across all GPUs and CPUs in the system and by providing tighter and more straightforward language integration for CUDA programmers.
- **Data access speed is maximized** by transparently migrating data towards the processor using it.

Unified Memory

Unification of memory spaces means that there is **no longer any need for explicit memory transfers** between host and device. Any allocation created in the managed memory space is automatically migrated to where it is needed.

A program allocates managed memory in one of two ways:

- via the `cudaMallocManaged()` routine, which is semantically similar to `cudaMalloc()`;
- or by defining a global `__managed__` variable, which is semantically similar to a `__device__` variable.

Unified Memory

Without UM:

```
__global__ void AplusB(int *ret, int a, int b) {  
    ret[threadIdx.x] = a + b + threadIdx.x;  
}  
  
int main() {  
    int *ret;  
    cudaMalloc(&ret, 1000 * sizeof(int));  
    AplusB<<< 1, 1000 >>>(ret, 10, 100);  
    int *host_ret = (int *)malloc(1000 * sizeof(int));  
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);  
    for(int i = 0; i < 1000; i++)  
        printf("%d: A+B = %d\n", i, host_ret[i]);  
    free(host_ret);  
    cudaFree(ret);  
    return 0;  
}
```

This first example combines two numbers together on the GPU with a per-thread ID and returns the values in an array. Without managed memory, both host- and device-side storage for the return values is required (`host_ret` and `ret` in the example), as is an explicit copy between the two using `cudaMemcpy()`.

Unified Memory

With UM:

```
__global__ void AplusB(int *ret, int a, int b) {  
    ret[threadIdx.x] = a + b + threadIdx.x;  
}  
  
int main() {  
    int *ret;  
    cudaMallocManaged(&ret, 1000 * sizeof(int));  
    AplusB<<< 1, 1000 >>>(ret, 10, 100);  
    cudaDeviceSynchronize();  
    for(int i = 0; i < 1000; i++)  
        printf("%d: A+B = %d\n", i, ret[i]);  
    cudaFree(ret);  
    return 0;  
}
```

The Unified Memory version of the program allows direct access of GPU data from the host. Notice the `cudaMallocManaged()` routine, which returns a pointer valid from both host and device code. This allows `ret` to be used without a separate `host_ret` copy, greatly simplifying and reducing the size of the program.

Unified Memory

With UM (simplified):

```
__device__ __managed__ int ret[1000];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    AplusB<<< 1, 1000 >>>(10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

Language integration allows direct reference of a GPU-declared `__managed__` variable and simplifies a program further when global variables are used.

Notice how in the non-managed example, the synchronous `cudaMemcpy()` routine is used both to synchronize the kernel (that is, to wait for it to finish running), and to transfer the data to the host. The Unified Memory examples do not call `cudaMemcpy()` and so require an explicit `cudaDeviceSynchronize()` before the host program can safely use the output from the GPU.

Unified Memory

Unified Memory attempts to **optimize memory performance** by migrating data towards the device where it is being accessed (that is, moving data to host memory if the CPU is accessing it and to device memory if the GPU will access it).

Data migration is transparent to a program. The system will try to place data in the location where it can most efficiently be accessed without violating coherency.

Unified Memory

Devices of compute capability lower than 6.x cannot allocate more managed memory than the physical size of GPU memory.

Devices of compute capability 6.x extend addressing mode to support **49-bit virtual addressing**. This is large enough to cover the 48-bit virtual address spaces of modern CPUs, as well as the GPU's own memory.

The large virtual address space and page faulting capability enable applications to access the entire system virtual memory, not limited by the physical memory size of any one processor. This means that **applications can oversubscribe the memory system**: in other words they can allocate, access, and share arrays larger than the total physical capacity of the system, enabling out-of-core processing of very large datasets.

`cudaMallocManaged` will not run out of memory as long as there is enough system memory available for the allocation.



Hardware implementation

CUDA architecture

Every CUDA device is a scalable array of **Streaming Multiprocessors (SMs)**, each one including N **cores** (as a matter of fact, they are just ALUs).

Each multiprocessor creates, manages, schedules, and executes threads in groups of N parallel threads (1 for each core) called **warps**.

The SM employs an architecture called

SIMT (single-instruction, multiple-thread)

The multiprocessor maps each thread to one core, and each thread executes independently with its own instruction address and register state.

Compute capability

The compute capability of a device is defined by a **major revision number** X and a **minor revision number** Y, and is denoted by the X.Y pair.

Devices with the same major revision number are of the same core architecture.

The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.

The major revision number is 7 for devices based on the *Volta* architecture, 6 for devices based on the *Pascal* architecture, 5 for devices based on the *Maxwell* architecture, 3 for devices based on the *Kepler* architecture, 2 for devices based on the *Fermi* architecture, and 1 for devices based on the *Tesla* architecture. The *Tesla* and *Fermi* architectures are no longer supported starting with CUDA 7.0 and 9.0, respectively.

Do not confuse the compute capability with the CUDA version, which is the version of the *CUDA software platform*.

CUDA architecture

For devices of compute capability 2.0 (2.1), an SM consists of:

- 32 (48) CUDA cores for arithmetic operations
- 4 (8) special function units for single-precision floating-point transcendental functions
- 2 warp schedulers

CUDA architecture

For devices of compute capability 6.0 (6.1, 6.2), an SM consists of:

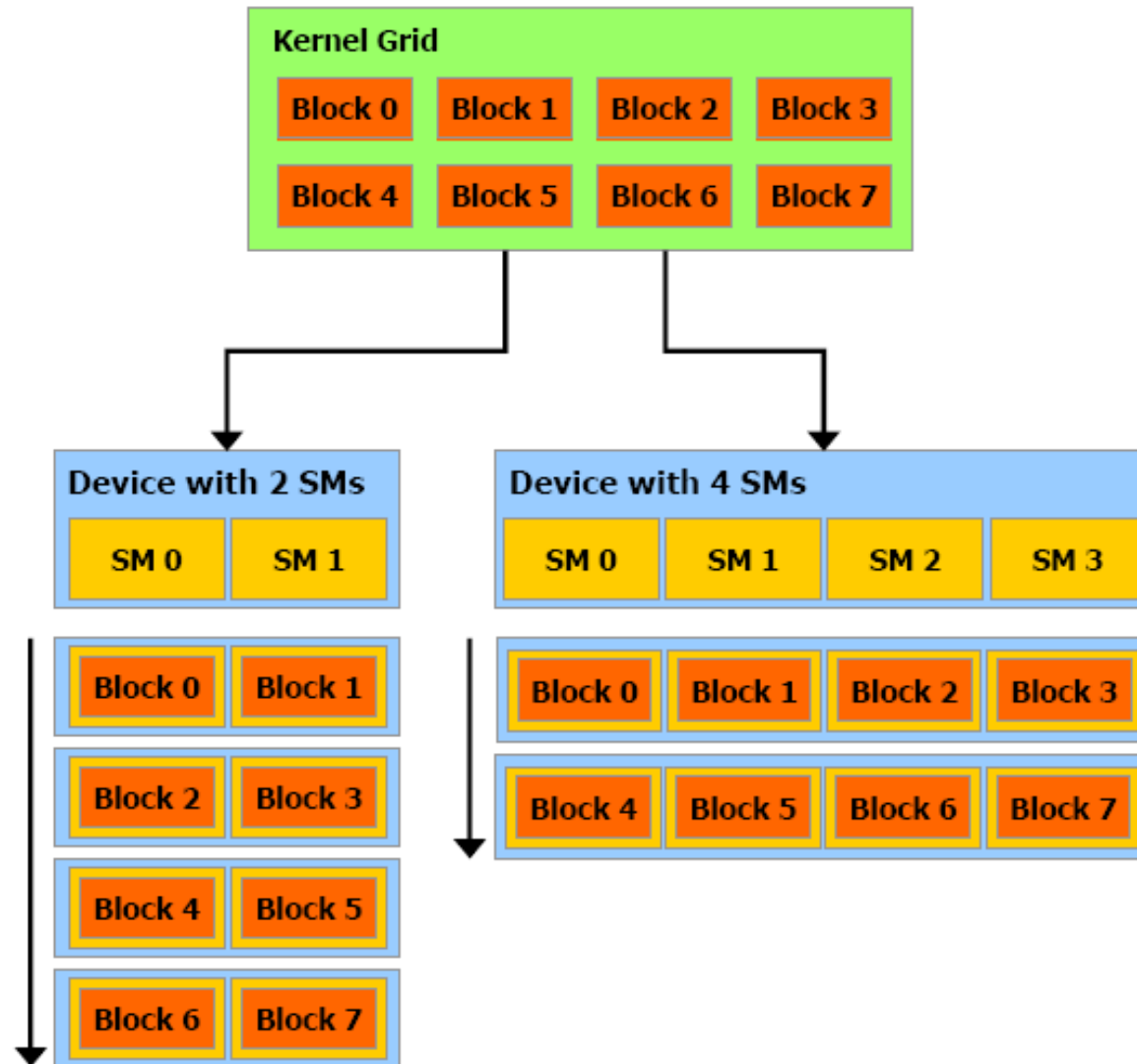
- 64 (128) CUDA cores for arithmetic operations
- 16 (32) special function units for single-precision floating-point transcendental functions
- 2 (4) warp schedulers

CUDA architecture

For devices of compute capability 7.x, an SM consists of:

- 64 FP32 cores for single-precision arithmetic operations
- 32 FP64 cores for double-precision arithmetic operations
- 64 INT32 cores for integer math
- 8 mixed-precision Tensor Cores for deep learning matrix arithmetic
- 16 special function units for single-precision floating-point transcendental functions
- 4 warp schedulers

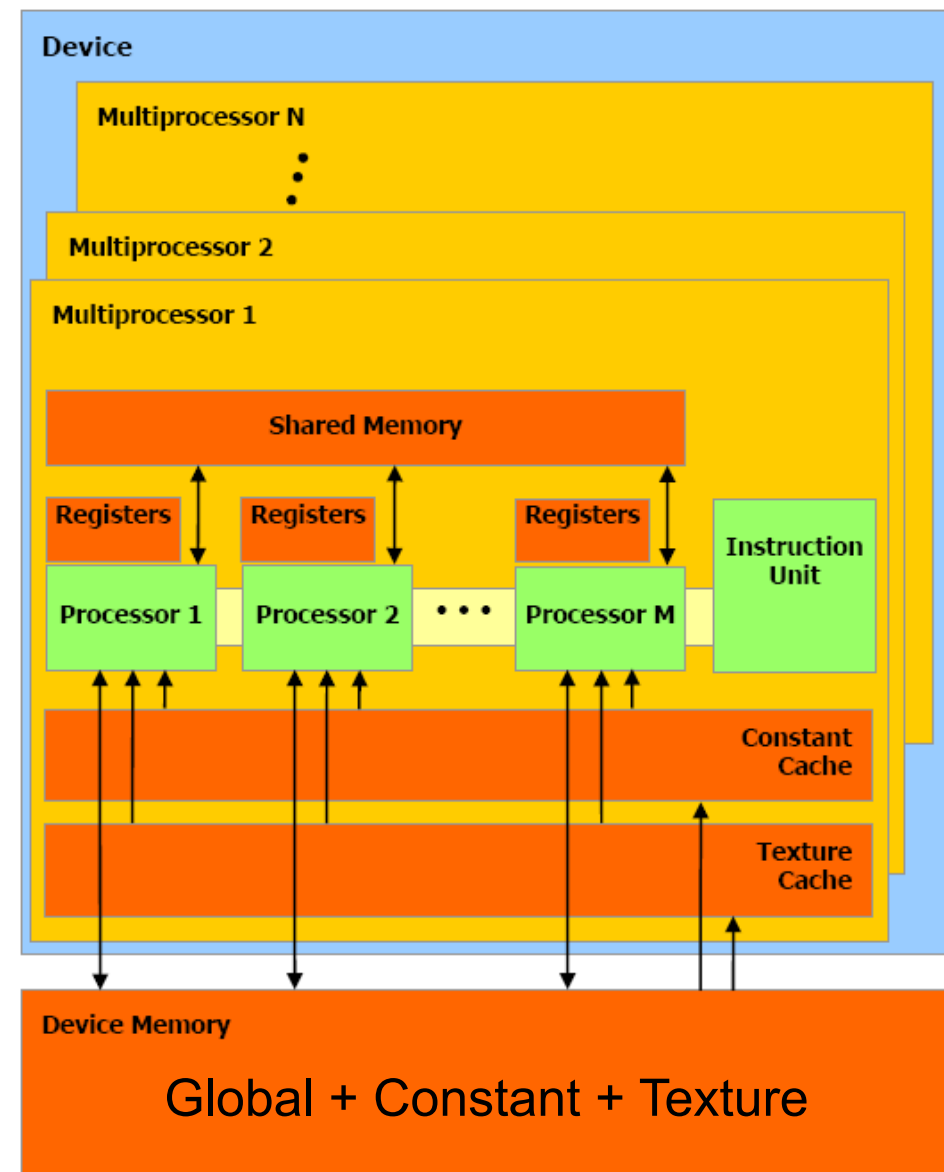
CUDA architecture



CUDA architecture

Each SM has on-chip memory of the four following types:

- ❑ One set of local **32-bit registers** per processor
- ❑ A parallel data cache or **shared memory** which is shared by all scalar processor cores and is where the shared memory space resides
- ❑ A read-only **constant cache** which is shared by all scalar processor cores and speeds up reads from the constant memory space, which is a read-only region of device memory
- ❑ A read-only **texture cache** which is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region of device memory





References

CUDA Zone

<https://developer.nvidia.com/cuda-zone>

NVIDIA CUDA - Programming Guide v12.0

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>